NASA Contractor Report 187571

ICASE Report No. 91-3

ICASE

A PARALLEL RENDERING ALGORITHM FOR MIMD ARCHITECTURES

Thomas W. Crockett
Tobias Orloff

NASA Contract No. NAS1-18605
June 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

A Parallel Rendering Algorithm for MIMD Architectures

Thomas W. Crockett
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center

Tobias Orloff
Great Northwestern Graphics

Abstract

Applications such as animation and scientific visualization demand high performance rendering of complex three dimensional scenes. To deliver the necessary rendering rates, highly parallel hardware architectures are required. The challenge is then to design algorithms and software which effectively use the hardware parallelism. This paper describes a rendering algorithm targeted to distributed memory MIMD architectures. For maximum performance, the algorithm exploits both object-level and pixel-level parallelism. The behavior of the algorithm is examined both analytically and experimentally. Its performance for large numbers of processors is found to be limited primarily by communication overheads. An experimental implementation for the Intel iPSC/860 shows increasing performance from 1 to 128 processors across a wide range of scene complexities. It is shown that minimal modifications to the algorithm will adapt it for use on shared memory architectures as well.

This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at ICASE.

Authors' addresses: Thomas W. Crockett, ICASE, M.S. 132C, NASA Langley Research Center, Hampton, VA 23665; Tobias Orloff, Great Northwestern Graphics, 119 North Fourth Street, Suite 206, Minneapolis, MN 55401.

Electronic mail: tom@icase.edu, orloff@poincare.geom.umn.edu.

1 Introduction

Applications such as real-time animation and scientific visualization demand high performance rendering of complex three-dimensional scenes. While the results achieved on current hardware have been impressive, major improvements in performance will require the use of highly parallel hardware and scalable parallel rendering algorithms. This paper describes one such rendering algorithm for MIMD architectures. Although the algorithm is designed for distributed memory message passing systems, straightforward modifications will adapt it for use in shared memory environments.

In the following section, we introduce the traditional rendering pipeline and consider the issues involved in parallelizing it. Next, we present our algorithm and give a theoretical analysis of its performance. We then describe an implementation on the Intel iPSC/860 [1] hypercube, and compare the experimental results with analytical predictions. Finally, we examine how the algorithm can be adapted for shared memory MIMD architectures.

2 The Rendering Problem

We assume that we are given a scene consisting of objects described as collections of 3D triangles, some light sources, and a viewpoint. The goal is to produce a 2D representation of the scene taking into account the lighting and perspective distortion (Fig. 1). For simplicity we assume the lights are all point light sources and the triangles possess only a diffuse coloring attribute. The addition of other material properties, such as specularity and texture, do not really affect the main structure of the algorithm.

There is now a fairly well established pipeline for the fast rendering of such three dimensional scenes [10]. The standard pipeline may be represented as shown in Figure 2. The exact sequence is not fixed, for example shading may be done after transforming (indeed, an essential portion of Phong shading must be done in the rasterizing step [3]), or clipping may be delayed until after the rasterization step.

One way to parallelize the rendering process is to map the various stages of the pipeline directly into hardware [2]. This approach has been very successful, and has been adopted by a number of graphics hardware vendors. But the ultimate performance attainable by directly exploiting the pipeline is limited by the number of stages in the pipe. To achieve a greater degree of parallelism, other strategies must be examined.

As is well known [11], there are three main steps in the rendering process which account for most of the computation time. These are

1. The floating point operations performed on objects, such as transforming, lighting, and clipping.

2. The rasterization of primitives transformed into screen coordinates.

3. Writing pixels to the frame buffer.

(We ignore here the problem of traversing the database prior to rendering.) The rendering time will be limited by the slowest of these three steps. Moreover, in current serial and pipelined hardware implementations, each of these three steps is operating at its limit [11]. Thus to obtain significant

[1] *iPSC*, *iPSC/2*, *iPSC/860*, and *i860* are trademarks of Intel Corporation.

improvements in performance, it is necessary to map the rendering pipeline onto a hardware architecture in which each of these three steps can be parallelized, preferably by replicating one basic type of processing element. We refer to parallel computations in step 1 as *object parallelism*, and in steps 2 and 3 as *image* or *pixel parallelism*. A system with a high degree of object parallelism is described by Torberg in [12]. A system with a high degree of pixel parallelism, the classic Pixel-Planes system of Fuchs and Poulton, is described in [6]. Finally, a system incorporating both object and pixel parallelism is described by Fuchs *et al.* in [7]. In all these cases, the algorithms for 3D rendering are mapped onto specific hardware, more or less constructed for that purpose. In our case, we map the rendering algorithm onto more general purpose parallel architectures. This allows us to experiment with the algorithm at a high level and with a high degree of flexibility. Once the critical performance parameters and tradeoffs are thoroughly understood, then special-purpose hardware can be designed to achieve maximum performance. As we will show, the algorithm described in this paper achieves both object and pixel parallelism, and will run on systems containing from 1 to p processors, where p is bounded by the number of scanlines. For an excellent discussion of the various approaches to object and pixel parallelization, see [11].

Besides exploiting both types of parallelism, a good algorithm must ensure that all large data structures are distributed among the processors without wasteful duplication. In our case there are two such structures: the list of triangles and the frame buffer. We distribute these structures evenly among the processors, allowing the algorithm to scale to more complex scenes and higher resolutions. Note that distributing the triangles corresponds to object parallelism, while distributing the frame buffer corresponds to pixel parallelism.

3 Algorithm Description

To describe the algorithm we first specify how the data structures are divided among the processors:

- The triangles are distributed evenly in round-robin fashion to all processors.
- The frame buffer is divided among the processors by horizontal stripes (Fig. 3).
- Small data structures, such as the lights and viewing parameters, are replicated on each processor.

The distribution of the frame buffer can be modified considerably without affecting the basic structure of the algorithm. Essentially all that is needed is a regular geometric division. We have implemented only a division into horizontal stripes, which seems appropriate for rendering into a frame buffer of size 1024 x 1024 using from 2 to 128 processors. The effect on performance of different splittings of the frame buffer is an interesting topic for further research.

With the above distribution of data, the following strategy is used:

- The shading, transforming, and clipping steps are performed by each processor on its local triangles.
- Before rasterizing a triangle, it is first transformed into screen coordinates, then split (if necessary) into trapezoids along local frame buffer boundaries (Fig. 4). Each trapezoid is then sent to the processor which owns the segment of the frame buffer in which it lies.
- Upon receiving a trapezoid, a given processor rasterizes it into its local frame buffer using a standard z-buffer algorithm[4] to eliminate hidden surfaces.

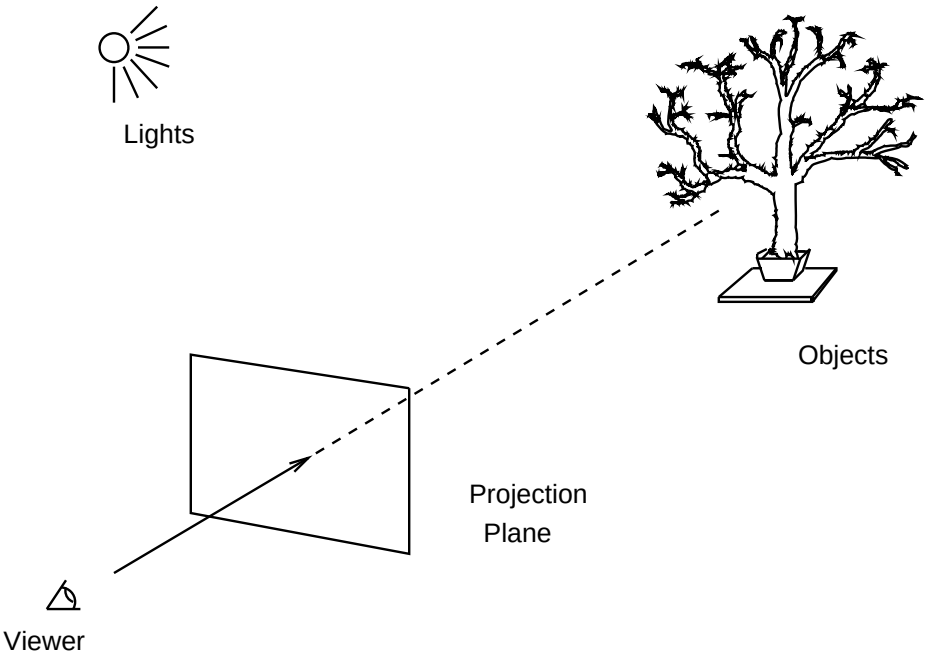

Figure 1: A scene is described by a collection of objects and light sources, with associated viewing parameters.

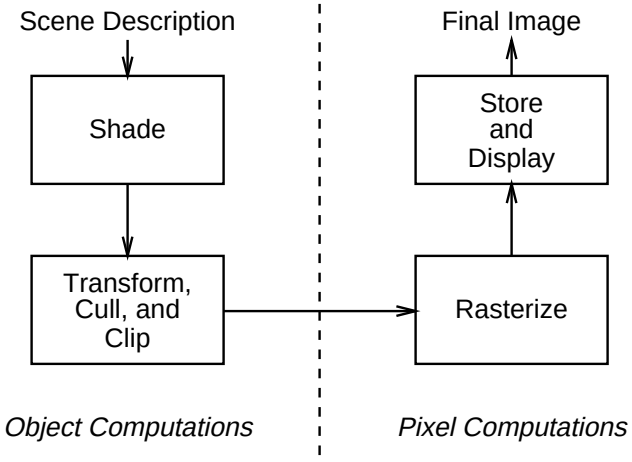

Figure 2: A typical rendering pipeline. The dotted line divides object-level and pixel-level operations.

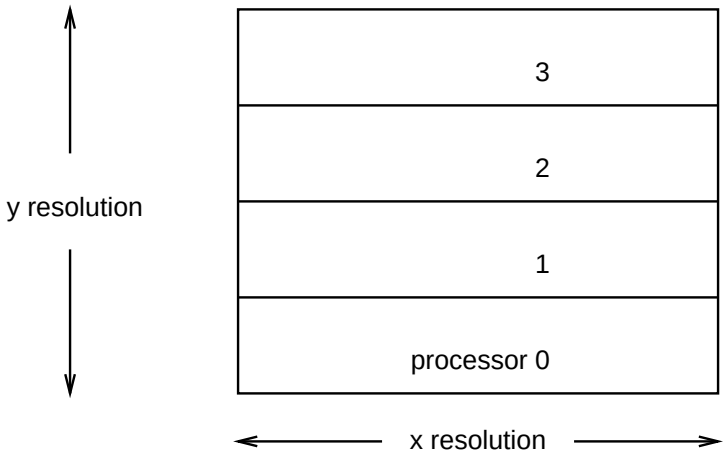

Figure 3: The frame buffer is distributed across processors by horizontal stripes.

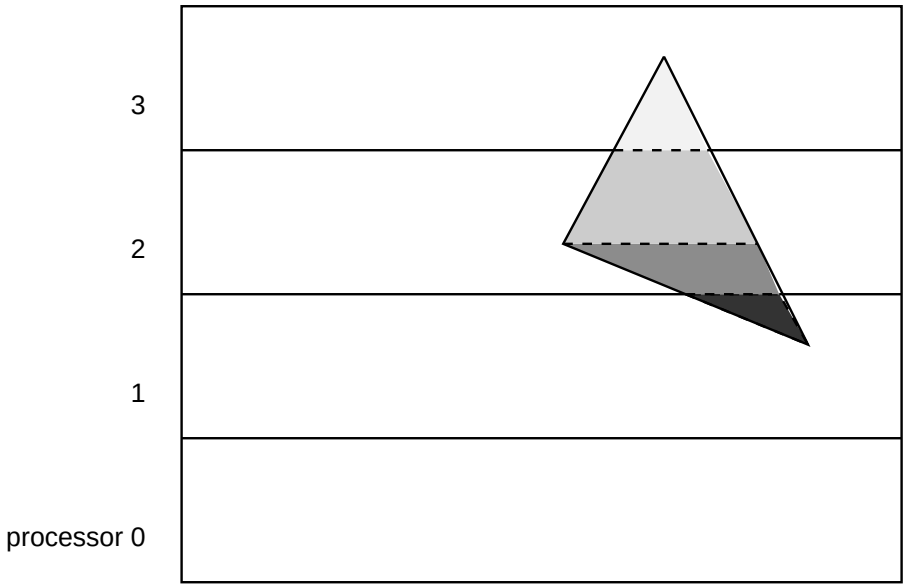

Figure 4: In general, triangles must be split at frame buffer boundaries and the pieces sent to the correct processors for rasterization.

For simplicity, triangles which lie fully within a single frame buffer segment and triangular pieces of split triangles are treated as degenerate trapezoids in which two of the vertices happen to be the same point. To reduce communication overhead, trapezoids destined for the same processor are buffered into larger messages before sending. The choice of buffer size, which can significantly affect performance, is discussed more fully later.

The algorithm may be summarized as follows. Each processor performs the loop:

```
Until done {

    If local triangles remain {
        Select a local triangle
        Shade the triangle
        Transform, back face cull, and clip
        Split into trapezoids
        Put the trapezoids into outgoing buffers
        When a buffer fills up, send its contents
        If this is the last local triangle {
            Send all non-empty buffers
        }
    }

    If incoming messages exist {
        For each incoming message {
            Rasterize all of the trapezoids in the message
        }
    }

}
```

To avoid having to store large numbers of trapezoids in memory, the algorithm alternates between splitting triangles into trapezoids and disposing of incoming trapezoids by rasterizing them into the frame buffer. It is not obvious what the proper balance is between these two activities. If a processor concentrates on rasterizing incoming trapezoids, it may starve other processors by not generating enough work to keep them busy. Alternatively, if incoming messages are not flushed quickly enough, message queues will fill up and outgoing buffers will be delayed. Experiments have shown that there is a slight advantage in processing at least a few triangles before checking for incoming data. Beyond that, the algorithm is relatively insensitive to this choice.

A significant feature of this algorithm is the absence of a synchronization point in the loop. Processors will start off with nearly the same number of triangles, but several factors will tend to unbalance the workload. First, the culling and clipping step requires a different number of operations for different triangles, and may cause triangles to be thrown away, or to be subdivided into several smaller triangles. Next, the time required for splitting into trapezoids varies with the orientation of the triangle and the number of frame buffer boundaries which are intersected. The number of trapezoids in turn affects the buffering and communication times. Similarly, varying numbers of incoming trapezoids, along with differences in their size and the results of z-buffer comparisons, will cause variations in the rasterization time.

These considerations suggest that any synchronization points in the loop will introduce significant amounts of idle time, since each iteration of the loop would be bound by the slowest processor. Instead, our strategy is to let individual processors proceed as asynchronously as possible. Of

course, some coordination is necessary to ensure that message buffers are correctly passed from one processor to the next. But the use of an asynchronous message-passing protocol, combined with dual send and receive buffers, has proven effective in minimizing idle time spent waiting for messages.

However, the lack of a synchronization point leads to difficulties in deciding when to exit the loop. Even after a given processor completes work on it local triangles, it has no way of determining by itself when it has received the last incoming message from another processor. We use the following algorithm to detect termination:

1. Each processor maintains a list of all other processors to which it sends trapezoids. We refer to these as *neighbors* of the sending processor. Note that a processor may be a neighbor of itself.

2. After the last local triangle is processed, a processor sends a *Last Trapezoid* (*LT*) message to each of its neighbors, indicating that there will be no more work forthcoming from that particular source. The message passing protocols must preserve message order so that LT messages do not precede the trapezoids to which they refer.

3. When a processor receives an LT message, it replies with a *Last Trapezoid Complete* (*LTC*) message. Receipt of an LTC message from a neighbor indicates that the neighbor has finished rasterizing all of the trapezoids sent to it. The processor records this fact.

4. When LTC messages are received from every neighbor, a processor knows that all of its neighbors have finished all of the work that it gave to them. The processor then produces a *Neighbors Complete* (*NC*) message which it sends to a specific processor, which we arbitrarily choose to be processor 0.

5. When processor 0 receives an NC message from every processor (including itself), it knows that each processor has finished all of the work sent to it by all of its neighbors, and that the local frame buffer segments now contain the final image. Processor 0 then broadcasts a *Global Completion* (*GC*) message to all processors. Receipt of a GC message notifies a processor that rendering is complete and that it should drop out of the loop.

From the time it generates LT messages for its neighbors until the time it receives a GC message, a processor must continue to check for incoming trapezoids and process them. Note that for a given number of processors p, the NC messages can be accumulated in time $O(\log p)$ using a parallel merge algorithm, rather than using the $O(p)$ method described above. Similarly, the GC broadcast can be done in time $O(\log p)$, or even $O(1)$ if the architecture directly supports broadcasts.

4 Performance Analysis

To analyze the performance of the algorithm we will break it down into the following steps:

- Shading, transforming, culling, and clipping.
- Splitting into trapezoids.
- Sending trapezoids.
- Rasterizing trapezoids.

- Storing pixel data.
- Wait time.
- Termination algorithm.

For each of these steps, we will break down the running time into a general *linear* part and an explicit *nonlinear* part. The linear component is that part which parallelizes perfectly, and thus will speed up linearly as the number of processors increases. The nonlinear component contains overheads which do not decrease linearly with increasing numbers of processors, and which therefore detract from perfect speedups. Before proceeding we introduce the following notation:

p = number of processors
n = number of triangles
y = height of the frame buffer (in scanlines)
h = average height of triangles (in pixels)
d = trapezoid buffer depth (in trapezoids)
τ = number of trapezoids generated per processor
υ = average number of trapezoids generated per neighbor

We assume that y and h are fixed, y is a multiple of p, $n \gg p$, and that the triangles comprising the scene are uniformly distributed with respect to our splitting of the frame buffer. Note that h is the average triangle height on the projection plane, rather than in world coordinates. The linear part of the running time is a term of the form

$$C\frac{n}{p} \tag{1}$$

where the constant C is machine- and scene-dependent, but independent of n, p, and d. This is the contribution to the running time that parallelizes perfectly. The nonlinear part of the running time will be everything else. We will attempt to determine this as explicitly as possible in terms of the above variables and machine dependent constants.

4.1 Shading, transforming, culling, and clipping

Since the triangles have been distributed evenly to the processors, and these operations may be performed independently on each triangle, this part of the algorithm contributes only a linear term to the running time.[2]

4.2 Splitting into trapezoids

Each triangle must first be split at its middle vertex (see Figure 4). Since this can be performed independently for each triangle, it contributes only a linear term to the running time. As a side effect, this split effectively doubles the number of triangles to $2n$ while reducing their average height to $h/2$. Next, there is a certain setup cost before actually dividing the triangle into trapezoids. Although this cost would not be incurred in a serial version of this algorithm, in the parallel version it still contributes only a linear term to the total running time. This cost may be regarded as part of the parallel overhead of the algorithm. Although we are not explicitly isolating the parallel

[2]Strictly speaking, back face culling and clipping can introduce local variations in workload which will detract from perfect speedup. But since we are assuming a uniform scene for purposes of analysis, we can ignore this effect. Similar variations can be introduced in the rasterization and z-buffer computations, and will likewise be ignored. In practice, the impact of these variations is scene-dependent.

overhead in our analysis, it does in fact contribute very little to the running time, so that the performance of the parallel algorithm running on one processor is virtually identical to that of a serial version.

A nonlinear contribution to the running time results from actually splitting the triangle. In loose terms, the more processors we have the more we must split the triangle, so that adding processors increases the number of trapezoids in the system. To quantify this, one easily computes that a triangle in the projection plane crosses a local frame buffer boundary line on average $hp/2y$ times. Since back face culling will, on average, eliminate half the original triangles, the number of resulting trapezoids per processor is

$$\tau \;=\; \frac{\frac{2n}{2}\left(\frac{hp}{2y}+1\right)}{p} \;=\; \frac{nh}{2y}+\frac{n}{p} \tag{2}$$

and the time to split n triangles among p processors is simply τt_{split}, where t_{split} is the time for one split. We can further analyze t_{split} by counting the arithmetic operations performed. The actual time will of course depend on the precise assembly code generated and the characteristics of the processor. In our current implementation, one split requires 15 integer adds and 10 integer compares.

4.3 Sending trapezoids

To a first approximation we assume that

- A single processor sending several messages must do so one at a time.
- Multiple processors can be sending simultaneously.
- A processor does not incur communication overheads for messages to itself.

The second assumption in particular is somewhat questionable—edge contention among competing sends can seriously impair message passing performance, as shown in [1]. This point will be discussed more fully in later sections.

Communication time can be divided into two independent parts, a fixed overhead, or *latency*, t_l, and a transfer cost t_t. The latency includes various software overheads and hardware delays, effects from network contention, etc. This is incurred on a per message basis. The transfer cost is just the inverse of the network bandwidth multiplied by the total number of bytes to be communicated.

A processor will, on average, generate $v = \tau/p$ trapezoids for each of p destinations, including itself. If t_b is the per-byte transfer cost and s is the size of a trapezoid in bytes, then

$$t_t = (p-1)vst_b \tag{3}$$

Taking into account buffering, the number of messages m generated by each processor is

$$m = (p-1)\left\lceil\frac{v}{d}\right\rceil \tag{4}$$

and the total time for sending trapezoids is simply

$$t_{send} \;=\; mt_l + t_t \;=\; (p-1)\left(\left\lceil\frac{v}{d}\right\rceil t_l + vst_b\right) \tag{5}$$

4.4 Rasterizing trapezoids

Since each pixel of each trapezoid is rasterized exactly once, and this work is split equally among the processors, it would appear that this part of the algorithm is linear. However, by splitting the triangles into trapezoids we incur an overhead for each trapezoid prior to rasterization. The rasterization step essentially consists of several applications of the Bresenham linear interpolation algorithm [5], once in the vertical direction and once per scanline in the horizontal direction. The overhead is incurred in the vertical application of the Bresenham algorithm, which must be performed for every trapezoid. Therefore the nonlinear contribution to the running time is τt_B, where t_B is the startup cost for the Bresenham algorithm. In terms of integer arithmetic operations, t_B is 5 divides, 10 multiplies, and 20 adds.

4.5 Storing pixels

The z-buffer compare and conditional store operations are perfectly distributed among the processors, so this computation contributes only a linear term to the running time.

4.6 Wait time

The rendering algorithm as viewed by a single processor consists of two distinct phases. During the first phase, the processor alternates between processing triangles and rasterizing trapezoids. If no trapezoids have arrived during a loop iteration, the processor can keep busy by processing more triangles during the subsequent iteration. In the second phase, all local triangles have been processed and the processor polls for incoming trapezoids until a Global Completion (GC) message arrives. During this second phase, the processor will be idle if trapezoids fail to arrive at least as fast as they can be rasterized. Furthermore, no processor can terminate until the slowest one has finished.

A precise treatment of this situation requires an excursion into queueing theory, which is beyond the scope of this paper. Instead, we present an argument based on the capacity of the communication network which approximates the observed performance of the algorithm. Our argument assumes that performance is communication bound, rather than compute bound. For purposes of exposition we will use a hypercube architecture. A similar analysis could be applied to other communication networks.

When a processor completes its last triangle, it must flush its partially filled trapezoid buffers. Since one of the goals of our algorithm is to conserve memory, we will assume that $d \leq v/2$, in which case outgoing buffers will contain on average $d/2$ trapezoids remaining to be sent. If we assume a uniform scene, then processors will reach this state at more or less the same time. Therefore the entire system contains $p(p-1)$ messages of average length $d/2$ which will be injected into the network at about the same time. Because of edge contention, these messages cannot all be sent simultaneously. For a hypercube of size $p = 2^k$ processors, the average distance a message must travel, and therefore the number of edges it ties up, is $kp/2(p-1)$. Since the total number of edges in a hypercube (assuming unidirectional communication) is $pk/2$ we have a bandwidth deficit of the order

$$\frac{p\,(p-1)\,\frac{d}{2}\frac{kp}{2(p-1)}}{\frac{pk}{2}} \;=\; \frac{pd}{2} \tag{6}$$

Thus wait time, t_{wait}, is roughly proportional to the shortage of communication capacity:

$$t_{wait} = \alpha\frac{pd}{2} \tag{7}$$

In a subsequent section, we will determine α empirically for a particular implementation.

4.7 Termination algorithm

The termination algorithm requires each pair of neighbors to exchange messages, followed by a global merge step and a broadcast. The time required for these last two operations depends on the architecture of the interconnection network. If we assume a hypercube, then

$$t_{quit} = 2\left[(p-1) + \log_2 p\right] t_l \tag{8}$$

There is no per-byte cost, since these messages are used only as signals and contain no data.[3]

4.8 Total time

Combining all of the above contributions, we find the total running time t for the algorithm:

$$t = C\frac{n}{p} + \tau t_{split} + t_{send} + \tau t_B + t_{wait} + t_{quit} \tag{9}$$

Substituting and rearranging, we get

$$t = C\frac{n}{p} + \tau\left(t_{split} + t_B\right) + (p-1)\, v s t_b + 2\left[(p-1) + \log_2 p\right] t_l + (p-1)\left\lceil\frac{v}{d}\right\rceil t_l + \alpha\frac{pd}{2} \tag{10}$$

In section 6 we present the experimental results for a particular implementation of this algorithm, and compare those results with predictions from the analytical model. First, we describe some pertinent details of our implementation.

5 iPSC/860 Implementation

We have implemented the above rendering algorithm in the C language on the Intel iPSC/2 and iPSC/860 hypercube computers. All of the experiments described below were performed on the latter system. The actual implementation differs slightly from the algorithm described above, in that shading calculations are pulled out of the main loop and done as a preprocessing step. (This is advantageous if a shaded scene will be displayed repeatedly using different viewing parameters.) Consequently, the rendering rates quoted below do not include the time for shading. Remember that the shading step parallelizes perfectly, and so would only improve the observed processor utilization. We should also note that the iPSC systems do not currently provide a graphical display device, so rendering rates do not reflect the final display step of the pipeline in Figure 2.[4]

Our sample implementation incorporates a standard scanline-based, z-buffered triangle renderer. The shading calculations take into account diffuse and ambient lighting components at the triangle vertices, and the rasterization process smoothly interpolates these values across the triangle. We use 8 bits for each of the red, green, and blue color channels, 24 bits for the z buffer, and pixel positions are maintained to a subpixel accuracy of one part in 64. The current implementation makes little attempt to optimize the graphics code for the i860 processor chip employed in the iPSC/860. The code is written entirely in a scalar (as opposed to vector) style, and no use has

[3] Actually, the message must at least convey its type, but we assume that all messages contain type information, which we include as part of the latency, t_l.

[4] Our current practice is to merge the finished contents of the local frame buffer segments into a file for later viewing offline. Our emphasis here is on the behavior of the parallel rendering algorithm, rather than on the use of the iPSC as a rendering engine.

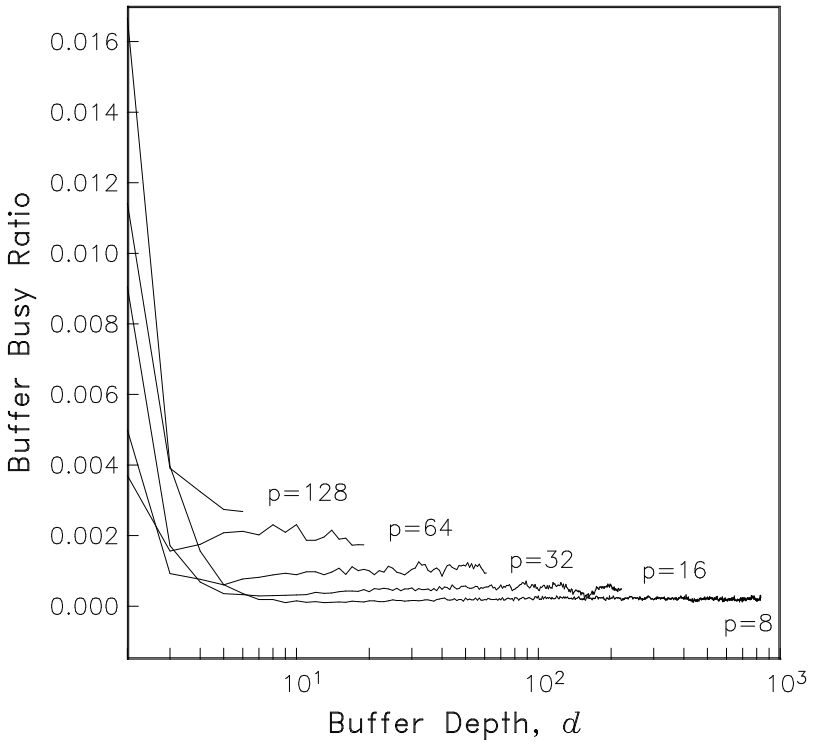

Figure 5: Buffer busy ratio *vs.* buffer depth.

been made of the built-in graphics features of the i860. In addition, the compilers available to us exploited few of the high performance capabilities of the i860, such as pipelining and dual instruction mode. With better compilers and some tuning, the performance of the graphics code should increase substantially, leaving communication and I/O as the limiting factors on rendering speed.

In contrast to the graphics computations, we have gone to some lengths to optimize message passing. The iPSC operating system provides asynchronous routines for both message sending and receiving, which can be used to overlap message transfers with other computations. We have taken advantage of these, in conjunction with a double-buffering scheme, to hide most of the overhead associated with message transfer time (t_t) as well as much of the edge contention delays. One measure of overlap is the number of times processors must wait when inserting trapezoids into outgoing buffers because the buffers are still busy from a previous send. Figure 5 shows this number expressed as the ratio of total buffer busy-waits to the total number of trapezoids generated across all of the processors. The values plotted are for buffer sizes ranging from 2 to $v/2$ with varying numbers of processors, using our standard test scene, described in the next section.[5] Each data point is the mean across five runs. It can be seen that the overlap strategy is very successful. In all cases, for $d \geq 3$, more than 99.5% of the trapezoids generated were able to be placed in buffers immediately.

[5]On the iPSC/860, different message passing protocols are employed for short messages ($\leq$ 100 bytes) *vs.* long messages ($>$ 100 bytes). Since our trapezoid data structure is 64 bytes long, buffers of depth 1 have different performance parameters than larger ones. For simplicity, we limit our analysis to buffer sizes ≥ 2.

6 Performance Results

In this section we present experimental results from the iPSC/860 implementation of our algorithm and compare them to predictions based on our performance model. Our standard test scene is composed of 100000 10 x 10 pixel triangles in random orientations (Fig. 6). This scene was chosen since it statistically approximates a uniform scene for purposes of comparison with the performance model. In all cases, we enable back face culling so that the number of triangles actually drawn is about half of the total. The scene is rendered with a frame buffer resolution of 512 x 512. The average triangle height on the projection plane as measured by the renderer is $h = 8.3$. To determine the effects of scene complexity, we modify the standard scene by varying the number of triangles while holding the triangle size, in pixels, constant. Unless otherwise noted, performance figures are mean values across five runs.

6.1 Sensitivity to buffer depth

As mentioned previously, the selection of buffer depth can have a significant impact on performance. Figure 7 shows scatterplots of rendering time *vs.* buffer depth for our standard scene, with p ranging from 8 to 128. (Because of memory requirements, a minimum of 8 processors are needed to render the standard scene.) Again, d ranges from 2 to $v/2$. The sensitivity to d can be readily understood in terms of the performance model. If d is small, then the ratio v/d in Equation 5 is large and the costs due to message latency are high. If d is large, latency is reduced but wait time due to network congestion increases (Eq. 7). For sufficiently large d, our algorithm is equivalent to a simpler two-phase version in which (1) all triangles are first split into trapezoids and the trapezoids are stored in memory, then (2) trapezoids are sent to their destinations and rasterized. It is clear from the performance model that this simpler algorithm not only wastes memory, but also maximizes edge contention by injecting all of the trapezoid data into the network at once. By using smaller buffer sizes and allowing splitting and rasterization to proceed together, our algorithm not only conserves memory, but spreads the communication load over a longer period of time.

For best performance, we would like to be able to predict an optimum buffer size, d_{opt}, without having to resort to a long series of test runs. If we know something about the scene, such as τ, or n and h, then the performance model can be used to determine a near-optimal value for d. (Recall that $v = \tau/p$.) If we take the derivative of Equation 10 with respect to d (ignoring the ceiling function) and solve for the minimum, we get

$$d_{opt} = \sqrt{\frac{2\,(p-1)\,v t_l}{\alpha p}} \tag{11}$$

The case where τ is unknown in discussed in Section 6.4 in the context of non-uniform scenes. We now turn our attention to values for t_l and α.

6.2 Message latency and wait time

Experimental measurements of message latency on the iPSC/860 have typically been done under carefully controlled test conditions in order to get consistent results. Because our algorithm is very dynamic, and because we include contributions due to buffer management, published values for message latency are not directly applicable. In addition, our simplistic analysis of wait time does not yield a value for the proportionality constant α. These considerations lead us to determine the values of t_l and α empirically. We recast Equation 10 as a function of d:

$$t(d) = C_0 + \frac{C_1}{d} + C_2 d \tag{12}$$

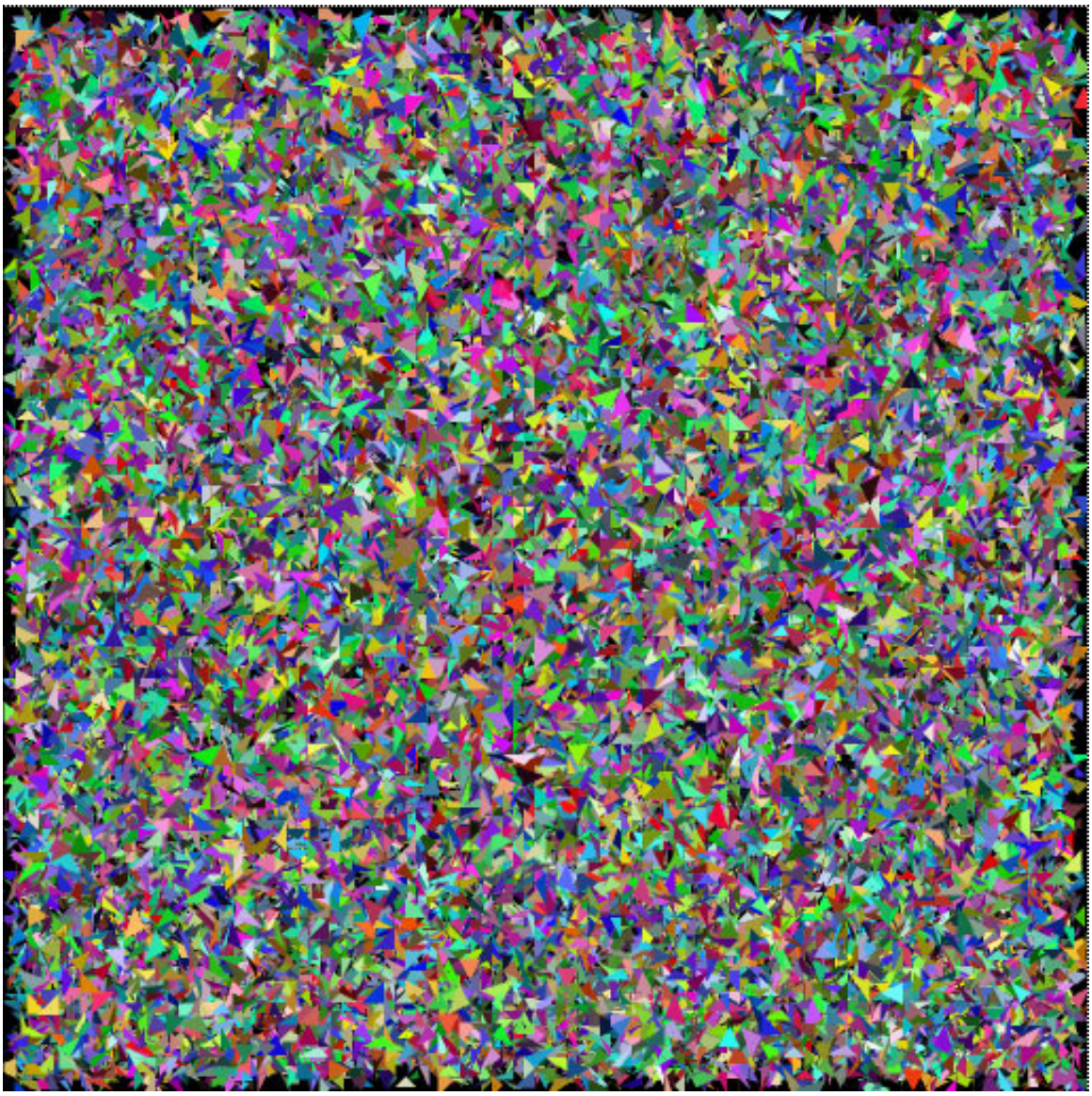

Figure 6: Standard test scene comprised of 100000 randomly oriented 10 x 10 pixel triangles. About half of the triangles have been eliminated by culling. $h = 8.3$.

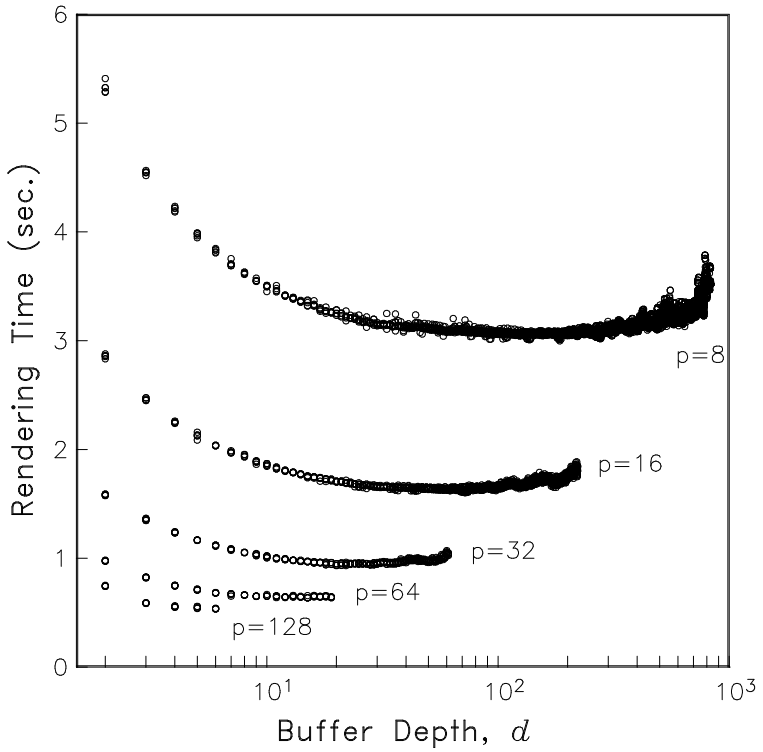

Figure 7: Execution time as a function of buffer depth for the standard test scene. $d = 2, \ldots, v/2$.

where

$$C_0 = C\frac{n}{p} + \tau\left(t_{split} + t_B\right) + 2\left[(p-1) + \log_2 p\right] t_{l_0} \quad (13)$$

$$C_1 = (p-1)\, v t_l \quad (14)$$

$$C_2 = \frac{\alpha p}{2} \quad (15)$$

For convenience, we again ignore the ceiling function. Because of the high degree of overlap achieved between communication and computation in our implementation, we have also dropped the term for data transfer time from Equation 3. Finally, we substitute t_{l_0} for t_l in the contribution from the termination algorithm to reflect the differing protocols for short and long messages. We can now do a least-squares fit using the data from our standard test scene to determine, approximately, the values of the coefficients C_0, C_1, and C_2, and then solve for t_l and α. The results are shown in Table 1.

The data suggest that t_l and α are not constants, but are instead functions of p, or more specifically, k, where $k = \log_2 p$. However, the limited sample size does not allow any firm conclusions to be drawn, and in the absence of a theoretical basis for determining the form of the functions, we have chosen to use the mean values.

6.3 Measured *vs.* predicted performance

To further explore the parallel performance of our algorithm and to validate the analytical model, we varied the complexity of the random triangle scene from 6250 to 200000 triangles in multiples of 2. For each scene, p ranged from the minimum allowed by memory requirements up to 128 in powers of 2, using the optimal buffer depth predicted by Equation 11 (Table 2). Figure 8a shows the

	Time in μs	
p	t_l	α
8	452	113
16	443	120
32	419	152
64	411	185
Mean	431	143

Table 1: Empirical values of message latency and wait time for the standard test scene.

	Predicted Optimal Buffer Depth					
p	*6250*	*12500*	*25000*	*50000*	*100000*	*200000*
2	69	98	138	–	–	–
4	43	60	85	121	–	–
8	23	33	47	66	94	–
16	12	18	25	35	50	71
32	7	9	13	19	27	38
64	4	5	7	10	15	21
128	2	3	4	6	9	12

Table 2: Predicted buffer sizes for several random triangle scenes.

observed rendering rates for each scene. The results show that performance continues to increase as processors are added, even for the smallest scene, although large numbers of processors are most effective for more complex scenes.

The usual measure of effectiveness of a parallel algorithm is *speedup*, defined as the time to execute a problem on a single processor divided by the time to execute it on p processors. In our case, only the smallest test scenes can be run on a single processor due to memory limitations, so traditional speedups cannot be computed directly. Instead, we normalize performance across scenes by comparing the rendering rates, instead of the execution time, and use these to estimate speedups.[6] We define the performance level for $p = 1$ to be the rendering rate of the largest test scene which would fit on a single processor, which was 4366 triangles/second for $n = 12500$. Table 3 shows speedups relative to this case. Speedups on large numbers of processors (64 and 128) are poor primarily due to communication costs (t_{send} and t_{wait}), which are the dominant overheads. As p decreases, the trapezoid costs (t_{split} and t_B) become the primary overheads, and speedups are reasonable on moderate numbers of processors (16 and 32). Figure 9 shows the relative contributions of the individual terms in the performance model for our standard test scene. On the plot, t_{split} and t_B have been combined into a single term, t_{trap}. Note that the contributions for t_{send}

[6] We consider our normalized speedup computations to be just estimates for two reasons: (1) As the density of the random triangle scenes increases, a larger proportion of the z-buffer comparisons will fail because pixels are obscured by other triangles which lie closer to the viewer. This results in a lower percentage of frame buffer stores and slightly reduced computational cost per pixel. (2) Because of the suspected effects of caching (described subsequently), execution times on small numbers of processors may not be directly comparable to those on larger numbers of processors. This effect could be mitigated by comparing performance at constant values of n/p, but the compensation is only partial since the size of the frame buffer segments is independent of the number of triangles.

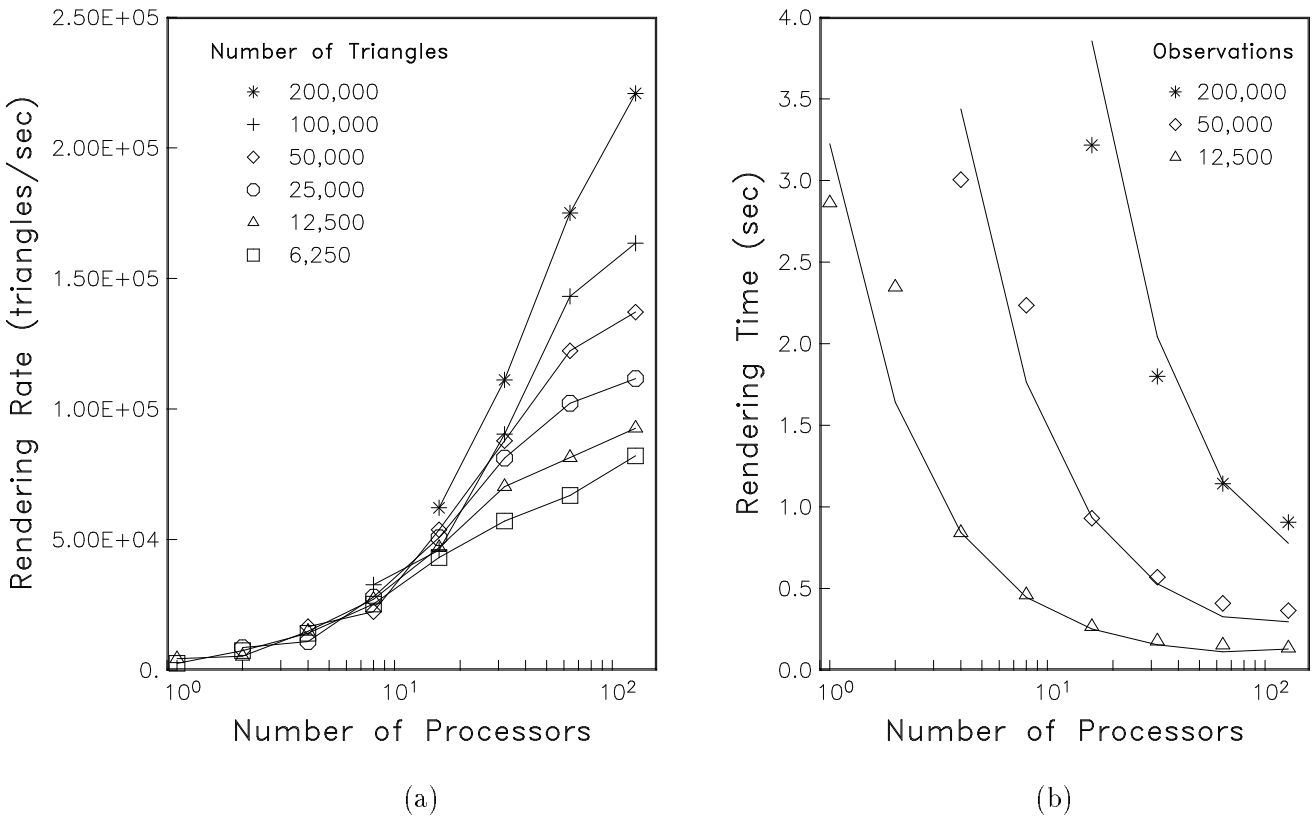

Figure 8: (a) Observed rendering rates for several random triangle scenes. (b) Observed and predicted rendering times for $n = 12500, 50000$, and 200000. Solid lines are the predicted performance.

and t_{wait} are roughly equal because the buffer size was chosen on the basis of Equation 11. The divergence of t_{send} and t_{wait} at large values of p illustrates the importance of the ceiling function in Equation 10, a contribution which was ignored in order to derive Equation 11.

We note in passing an interesting phenomenon in the speedup data. At certain points in the table (shown in bold type), dramatic increases in performance are observed from one value of p to the next. Since these points occur at fixed values of n/p, we conjecture that they are due to caching on the i860 processor. As p increases, the size of several data structures (triangles, frame buffer segment, message buffers) decreases, which may result in better cache hit ratios.

In Figure 8b, we compare the observed and predicted performance of several test scenes. To predict performance using our model, we must first determine the value of the scene-dependent constant C. This is done by taking the observed rendering time on some number of processors p and solving Equation 10 for C. We have chosen the entries lying along the boldface diagonal in Table 3 as the points at which to solve for C (points of constant n/p). We also need values for t_{split}, t_B, and t_{l_0}. Based on the operation counts from Section 4 and timing information from [8, 9], we estimate that $t_{split} = 2.500$ μs and $t_B = 13.375$ μs. Since communication in the termination algorithm uses synchronous (non-overlapped) message passing routines and incurs very little overhead beyond the actual message transmission, we use published latency data [1] to set $t_{l_0} = 75$ μs. As the plot shows, our model successfully predicts the general performance trends. Some discrepancies occur for small p where the suspected caching perturbations occur, and the model underestimates slightly the overheads at large p. This lends credence to our previous observation that α is an increasing

	Speedup					
p	*6250*	*12500*	*25000*	*50000*	*100000*	*200000*
1	0.6	1.0	–	–	–	–
2	**1.7**	1.2	2.0	–	–	–
4	3.2	**3.4**	2.5	3.8	–	–
8	5.8	6.2	**6.4**	5.1	7.5	–
16	9.9	10.7	11.6	**12.3**	10.5	14.2
32	13.1	16.1	18.6	20.1	**20.7**	25.5
64	15.3	18.6	23.4	28.0	32.8	40.1
128	18.8	21.2	25.6	31.4	37.4	50.6

Table 3: Speedup estimates derived from observed rendering rates. Boldface entries indicate unexpectedly large performance increases.

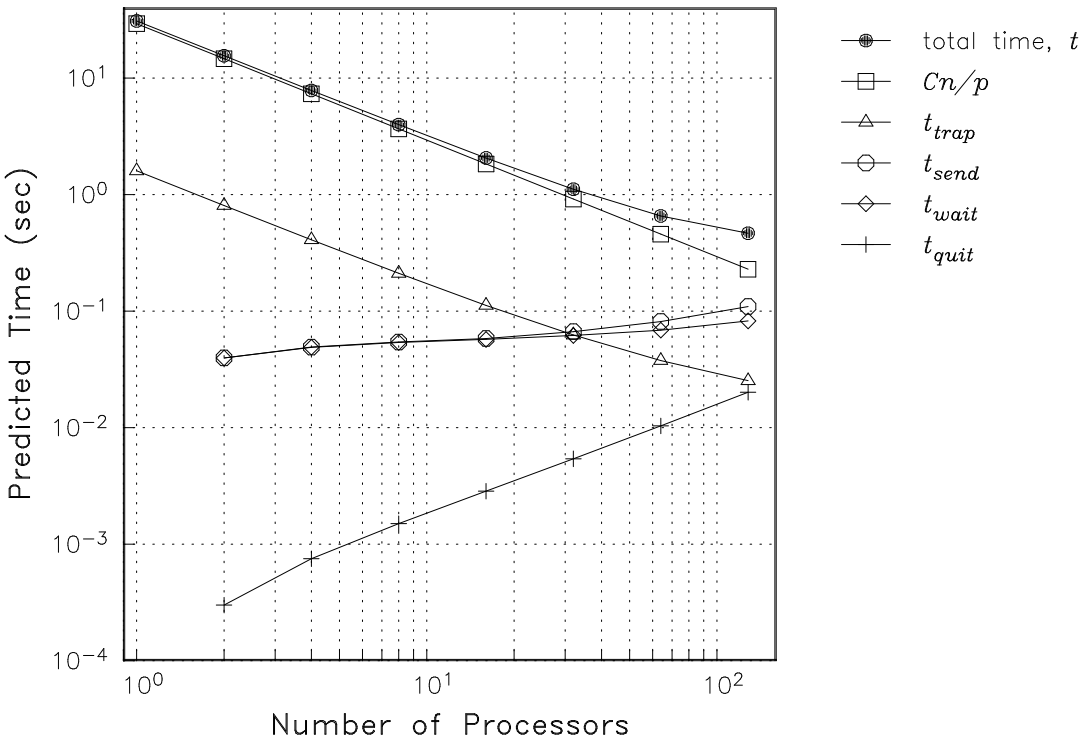

Figure 9: Predicted contributions of individual components of the performance model for our standard test scene. $t_{trap} = \tau(t_{split} + t_B)$.

function of k, although other terms may be involved as well.

6.4 Performance on non-uniform scenes

Although our random test scene is useful for analyzing our algorithm, it is not a very representative application. To obtain a better feel for performance on more realistic scenes, we ran experiments with two additional test cases, shown in Figures 10–11. The first scene, which we will refer to as *Plato*, contains a large number of small triangles with the density of triangles varying from place to place in the scene. The second scene, designated *LDEF*, contains a wide range of triangle sizes, which is very effective at desynchronizing the processors because of differences in rasterization time. Both scenes were rendered at a resolution of 512 x 512.

The first issue we address is that of picking a buffer size. Figure 12 shows rendering time as a function of buffer size, where d varies from 2 to $1.25v$. In contrast to the uniform scene shown in Figure 7, the optimal buffer sizes for both of these scenes occur at much larger values of d. Thus Equation 11 is not applicable because the processors are much farther out of sync and the final buffer flushes are spread out in time, reducing the effect of t_{wait}. In the absence of an analytical prediction for a good buffer size, we note that $v/2$ works well in many cases. Other experiments have shown that for small values of n/p, increasing the buffer depth to around v offers additional performance gains, a trend hinted at in Figure 12.

If τ, and hence v, are unknown, then the best we can do is hazard a guess. A buffer depth of 10-100 seems like a good starting point since it reduces latency costs by one to two orders of magnitude. As a rule, buffer depth should decrease with increasing p. If a scene will be rendered repeatedly with minor changes in the viewing parameters from frame to frame, as in an animated sequence, then the renderer can automatically adjust the buffer size. For the first frame an initial guess is needed. For subsequent frames, the observed value of τ from the previous frame is used to derive a better guess for d.

Figure 13 shows rendering rates for the LDEF and Plato scenes using a buffer depth of $v/2$. Because of memory limitations, neither of these scenes could be rendered with a single processor at 512 x 512 resolution. Hence we have no single-processor data on which to base speedup estimates. Examination of the available data shows that processor utilization is best for p of 16–32 or less, consistent with the previous results for scenes of these sizes. Note that performance of the Plato scene peaks out at 64 processors and then declines as the communication overhead becomes dominant. Careful choice of buffer sizes can boost the Plato performance on 64 and 128 nodes to about 125000 triangles/second.

7 Considerations for Shared Memory Architectures

Although the algorithm described in Section 3 was designed specifically for distributed memory machines, it can be readily adapted for shared memory architectures. We assume that viable shared memory systems would support an efficient mechanism for implementing critical sections on shared variables. Given this, the basic structure of the algorithm remains the same, with interprocessor communication taking place through shared data structures rather than with messages.

Instead of partitioning the triangles in round-robin fashion and assigning them to particular processors, they are placed in a shared list or array. When a processor needs a triangle to work on, it grabs the next one from the list, in typical self-scheduled fashion. The overhead for fetching the triangle is the time it takes to lock the list index variable, read the current value, increment it, and unlock it. Presumably this can be done in a few instruction times given suitable architectural support. Some wait time may be incurred if another processor already has the variable locked. This

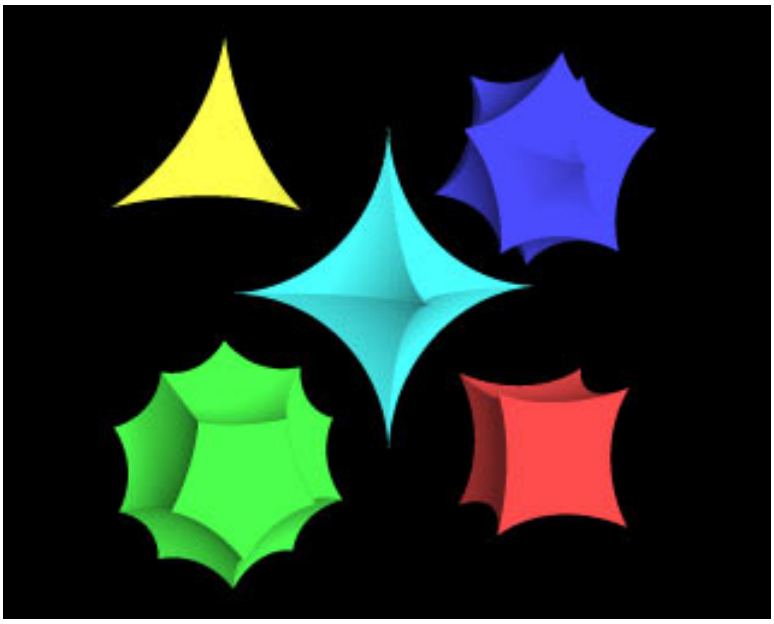

Figure 10: The five hyperbolic Platonic solids. $n = 59276$, $h = 3.1$.

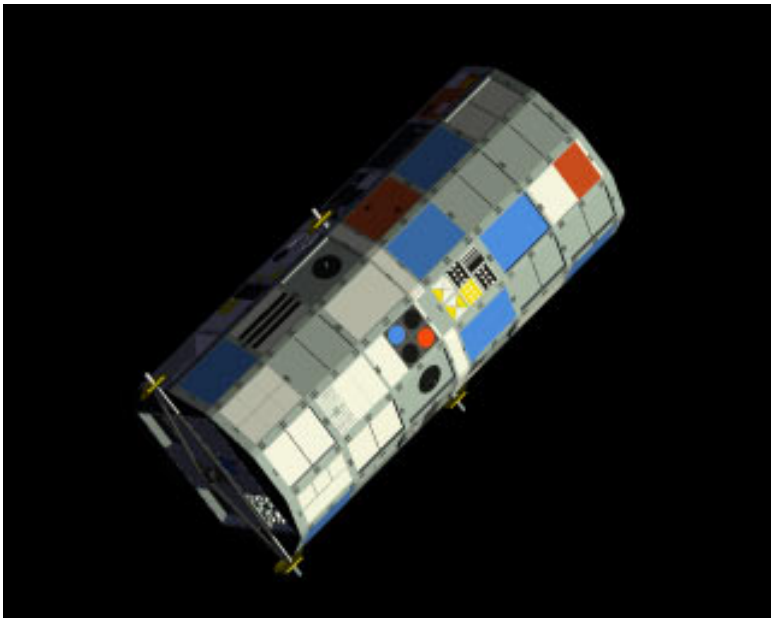

Figure 11: NASA's Long Duration Exposure Facility. $n = 17726$, $h = 6.2$.

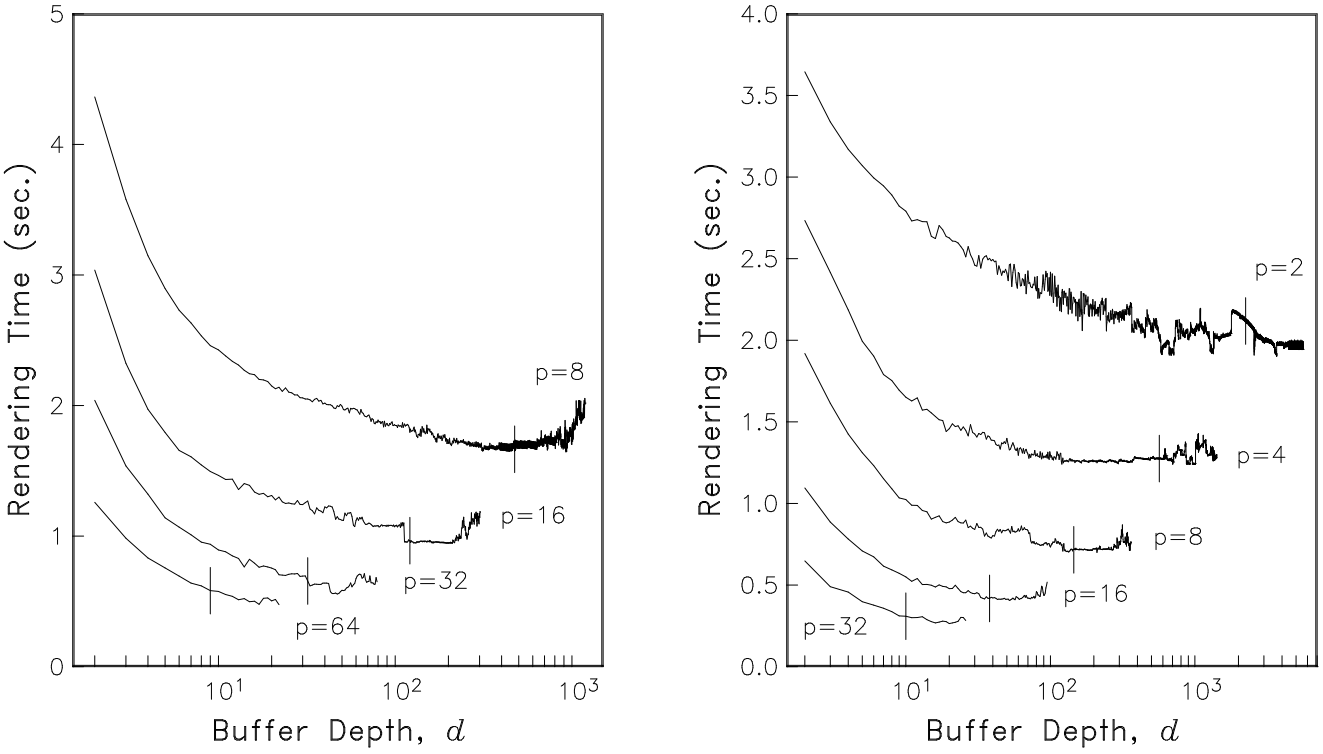

Figure 12: Execution time as a function of buffer depth for the (a) Plato and (b) LDEF scenes. d ranges from 2 to $1.25v$. The vertical bars indicate $d = v/2$.

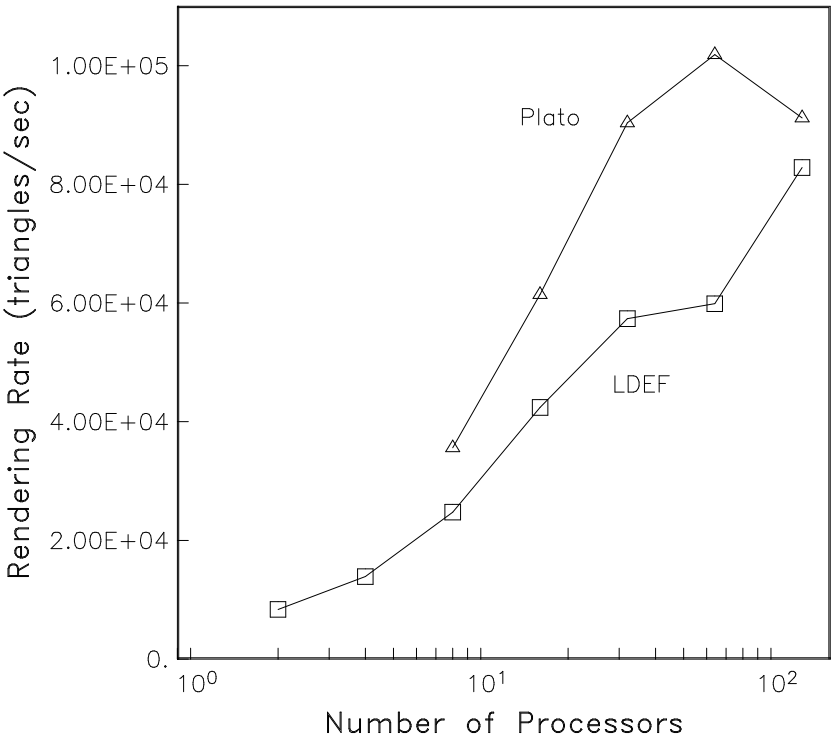

Figure 13: Rendering rates for the LDEF and Plato scenes.

should not be a problem for moderate numbers of processors, since the time to process a triangle would be much larger than the time required to fetch and update the list index.

The other major shared data structure is the frame buffer. A naive approach would be to let processors rasterize triangles directly into the frame buffer after transforming them into screen coordinates. But since many processors would be doing this simultaneously, there would be memory conflicts when triangles overlapped other triangles in the frame buffer. A poor solution would be to lock the entire frame buffer for the duration of the rasterization step, but that would effectively serialize the rasterization phase of the computation. A better solution is to partition the frame buffer into p segments. Then triangles could be split into trapezoids as in our original algorithm. But instead of sending the trapezoids to other processors, they would be placed on a shared list of trapezoids needing to be rasterized. There would be one trapezoid list per frame buffer segment. After processing one or more triangles, a processor would grab an unlocked frame buffer segment and process all of the outstanding trapezoids queued for that segment. Because there are as many segments as there are processors, at least one will always be unlocked. By not tying frame buffer segments to particular processors, load balancing will be automatic and performance should be better than the distributed memory version of the algorithm. As before, the overhead for maintaining the trapezoids lists and locking and unlocking frame buffer segments should be small compared to the cost of the rasterization computations.

Thus, the shared memory version of the algorithm becomes:

```
Until done

    If triangles remain
        Select the next triangle
        Shade the triangle
        Transform, back face cull, and clip
        Split into trapezoids
        Insert the trapezoids onto the trapezoid lists

    Find an unlocked frame buffer segment with outstanding
      trapezoids (if any)
        Rasterize all of the trapezoids in that list

Continue
```

Termination of the algorithm is also simpler in the shared memory version. Each processor must certify when it has finished working on its last triangle. This occurs when a processor checks for the next triangle and none remain. After all processors have finished their last triangle, then when all of the trapezoid lists become empty and all of the frame buffer segments are unlocked, rendering is complete.

Modification of the performance model for the shared memory algorithm is straightforward. An additional nonlinear term is needed to model contention for the triangle list index variable. Message passing terms in the distributed memory model are replaced with terms which reflect the time needed to update the trapezoid lists (including contention) and to search for unlocked frame buffer segments with outstanding trapezoids.

8 Conclusion

In this paper we have described a parallel rendering algorithm for MIMD computer architectures. The algorithm is attractive for its exploitation of both object and pixel level parallelism. We have given a theoretical analysis of its performance on distributed memory, message passing systems, and compared this with an actual implementation on the Intel iPSC/860 hypercube. Our results show that the algorithm is a viable means of achieving a highly parallel renderer. Scalability is limited primarily by communication costs, which increase as a function of the number of processors. Expected improvements in communication speed and optimization of the transformation and rasterization software will allow this algorithm to compete favorably with other high-performance rendering systems.

Acknowledgments

We would like to thank the Institute for Parallel Computation at the University of Virginia, the Mathematical Sciences Section at Oak Ridge National Laboratory, and the NAS Systems Division at NASA Ames Research Center for providing access to their iPSC computers at various stages during the development and testing of our algorithm. We would also like to thank Shahid Bokhari for sharing his expertise during many helpful discussions, and Harry Jordan for providing valuable assistance in refining the performance model. The geometric database for the LDEF scene was provided courtesy of the Flight Software and Graphics Branch at NASA Langley Research Center, and Computer Sciences Corporation.

References

[1] Bokhari, Shahid. Communication Overhead on the Intel iPSC-860 Hypercube. ICASE Interim Report 10 (NASA CR 182055), Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Va., May 1990.

[2] Clark, J. The geometry engine: a VLSI geometry system for graphics. *Computer Graphics*, Vol. 16, No. 3, July 1982, 127-133.

[3] Foley, J. D., van Dam, A., Feiner, S.K., and Hughes, J.F. *Computer Graphics: Principles and Practice.* 2nd ed., Addison-Wesley, Reading, Mass., 1990, 738-739.

[4] Foley *et al. op. cit.*, 668-672.

[5] Foley *et al. op. cit.*, 71-81.

[6] Fuchs, H., and Poulton, J. Pixel-Planes: a VLSI-oriented design for a raster graphics engine. *VLSI Design*, Vol. 2, No. 3, Q3 1981, 20-28.

[7] Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Isreal, L. Pixel-Planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics*, Vol. 23, No. 3, July 1989, 79-88.

[8] *i860 64-Bit Microprocessor Data Sheet.* Intel Corporation, Santa Clara, CA, 1989.

[9] *i860 64-Bit Microprocessor Programmer's Reference Manual.* Intel Corporation, Santa Clara, CA, 1990.

[10] Molnar, S., and Fuchs, H. Advanced raster graphics architecture. In Foley *et al., op. cit.*, 866-873.

[11] Molnar, S., and Fuchs, H. *op. cit.*, 855-923.

[12] Torberg, J. G. A parallel processor architecture for graphics arithmetic operations. *Computer Graphics*, Vol. 21, No. 4, July 1987, 197-204.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1991	Contractor Report

4. TITLE AND SUBTITLE
A PARALLEL RENDERING ALGORITHM
FOR MIMD ARCHITECTURES

5. FUNDING NUMBERS
C NAS1-18605
WU 505-90-52-01

6. AUTHOR(S)
Thomas W. Crockett
Tobias Orloff

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 91-3

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-187571
ICASE Report No. 91-3

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
To be submitted to Concurrency: Practice and Experience

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Applications such as animation and scientific visualization demand high performance rendering of complex three dimensional scenes. To deliver the necessary rendering rates, highly parallel hardware architectures are required. The challenge is then to design algorithms and software which effectively use the hardware parallelism. This paper describes a rendering algorithm targeted to distributed memory MIMD architectures. For maximum performance, the algorithm exploits both object-level and pixel-level parallelism. The behavior of the algorithm is examined both analytically and experimentally. Its performance for large numbers of processors is found to be limited primarily by communication overheads. An experimental implementation for the Intel iPSC/860 shows increasing performance from 1 to 128 processors across a wide range of scene complexities. It is shown that minimal modifications to the algorithm will adapt it for use on shared memory architectures as well.

14. SUBJECT TERMS	15. NUMBER OF PAGES
3D graphics; rendering; parallel algorithms; multiprocessors; performance analysis	25
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		